Computer Languages, Systems & Structures 44 (2015) 355–365

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Supporting comprehensible presentation of clone candidates through two-dimensional maximisation

Viktória Fördős [a], Melinda Tóth [b,*]

[a] Erlang Solutions Ltd., Hungary
[b] Eötvös Loránd University & ELTE-Soft Nonprofit Ltd., Hungary

ARTICLE INFO

Article history:
Received 20 January 2015
Received in revised form
8 August 2015
Accepted 26 September 2015
Available online 9 October 2015

MSC:
68N30

Keywords:
Code clones
Comprehensible result presentation
Grouping
Maximum clique

ABSTRACT

Duplicated code detection has been an active research field for several decades. Although many algorithms have been proposed, only a few researches have focussed on the comprehensive presentation of the detected clones. During the evaluation of clone detectors developed by the authors, it was observed that the results of the clone detectors were hard to comprehend. Therefore, in this paper a broadly suitable grouping method with which clone pairs can be easily grouped together to provide a more compact result is presented. The grouping algorithm is examined and a more precise revised algorithm is proposed to present all of the candidates to the user.

1. Introduction

The existence of code clones – mostly the results of "copy&paste programming" – can make software more prone to bugs and inconsistencies during maintenance. Thus, programmers should at least be aware of the existing clones in the software. Considering legacy code, the manual identification of clones is a tough challenge for an IT team. Fortunately, clone detectors [1] come to the rescue.

At the first sight the challenge of identifying code clones in software ends by the successful application of the duplicated code detectors, however, it is not always true. The benefit gained from the reported code clones greatly depends on whether the users are capable of exploiting the results. If the users are swamped with either irrelevant details or huge number of clones, they spend too much time on the analysis and it is likely that they oversee the most relevant and important clones, for instance, the code fragment duplicated several times.

When we evaluated our duplicated code detectors using legacy code [2–4], we found ourself in a situation that analysing the result is almost as hard as elaborating a new algorithm. Our algorithms result clone pairs, and in the case of legacy code the population of the detected clone pairs was such enormously large that we could not grasp the result. Moreover, as more than half of the clone pairs shared code fragments with the others, we analysed nearly the same clone again and again. In conclusion, it turned out that presenting clones as pairs made the result hard to understand, even though the result is

* Corresponding author.
E-mail addresses: viktoria.fordos@erlang-solutions.com (V. Fördős), tothmelinda@elte.hu (M. Tóth).

http://dx.doi.org/10.1016/j.cl.2015.09.004
1477-8424/

valuable and useful. We also noticed that grasping clone groups is much easier thanks to the compactness of the result. Yet, what can be done if the algorithms result in pairs?

In paper [5] a general solution to this problem was proposed. Nevertheless, the solution is not perfect for all the goals. If the goal is to efficiently eliminate the code fragments having the utmost clones with the least effort, the algorithm proposed in this paper is said to be the best. (It is the dominant opinion of programmers participating in our experiment.) The algorithm presented in this paper revises the general solution [5]. It is a language independent grouping method that reveals all of the possible groups of clones that cannot be further extended – either by including more code to each clone member or by adding new clone members to existing groups. It is not a duplicated code detector, it works with the result of the detectors – set of clone pairs – to provide a comprehensible and easily usable result by refining, reorganising the clone pairs.

2. Related work and background

Tools to support the detection of duplicated code fragments may differ in the used detection algorithm, the type/level of found clones, the presentation of the result, the targeted programming language, etc. These parameters are not fully orthogonal. The used detection algorithm can determine the presentation of code clones as well. For instance, when a suffix-tree based lexical approach is used the detection easily returns group of clones. Although, when a metric based pairwise comparison is used for detection, the result of the algorithm is usually pairs of clones. Returning hundreds of pairs makes the comprehension and processing of clones hard.

In this paper we introduce a general algorithm to form groups from clone pairs resulted by any kind of detection algorithm. Our approach is language independent, it only requires an *isClone* relation (Section 2.1.2) that describes whether two entities can be considered as a clone.

We have evaluated our approach in the context of the Erlang programming language and have used the tool called Clone IdentifiErl (Section 2.1.1). However, we would like to emphasise here that our approach does not restrict the programming language in which the clones were implemented and the detection algorithm used to retrieve the clone pairs.

In this section we first introduce some clone detector tools in general (Section 2.1) and in terms of the presentation of clones as well (Section 2.2). Finally in Section 2.3 we provide the background of our work. We give an overview of our initial approach to form clone groups, which was presented in paper [5]. The main contribution of this paper is the grouping algorithm presented in Section 3 which is a refinement of this initial approach.

2.1. Clone detectors

Clone detection is an active field of research, a large number of prominent research have been successfully carried out. The paper [1] gives an overview of the clone detection processes in general, and also provides a description of clone detection techniques (textual, lexical, syntactic and semantic approaches) and tools. They evaluated the tools based on some predefined editing scenarios. This study aimed to help the users in selecting the appropriate tool for their needs.

Another paper [6] focuses on the precision of six different tools that are using different techniques. They have been evaluated by the tools on C and Java programs.

There are some clone detectors, such as CCFinder [7], that aim to implement efficient and scalable algorithms. Thus, they have been successfully applied on industrial-size software as well. CCFinder also offers transformation rules to remove and eliminate certain kinds of clone instances.

Several clone detectors have been developed for the mainstream programming languages, while for functional programming languages only a few exist [8,9] adopting broadly usable, general duplicated code detector algorithms. That is a problem as nowadays there is a continuously increasing interest towards functional programming languages. Just consider that anonymous functions are available even in .NET and JAVA.

Considering soft-real time, fault-tolerant, large-scale software with high availability a new star is rising – the Erlang programming language [10]. It drives the telecom industry (e.g. Ericsson), large betting portals (e.g. Bet365), inspired Microsoft to introduce the Orleans framework [11] and also frequently used in FP7 research projects (e.g. ParaPhrase [12], RELEASE [13]). Erlang is a dynamically typed, concurrent, distributed, functional programming language. The most important language elements are the functions that are built up from function clauses. Each function clause contains at least one expression (called a *top-level expression*) or a sequence of top-level expressions, which usually corresponds to the smallest code clone.

Not surprisingly, a huge amount of legacy code exists written in Erlang, which likely contain code clones. Unfortunately, the difference between the functional and the object oriented paradigms is so significant that general algorithms cannot accommodate themselves and do not reveal all the clones in Erlang programs. To provide the complete visibility of the clones specialised algorithms [2–4] for Erlang were proposed in previous work by the authors. Our algorithms take into account the domain-specific knowledge of the language to improve their results. This knowledge is provided by RefactorErl [14,15] that is a source code analysis and transformation tool for Erlang.